# FORTH ASSEMBLER
For 6502 Microprocessor

## OVERVIEW

Forth is provided with a machine language assembler to create execution proceedures that would be time inefficient, if written as colon-definitions. It is intended that "code" be written similarly to high level, for clarity of expression. Functions may be written first in high-level, tested, and then re-coded into assembly, with a minimum of restructuring.

## THE ASSEMBLY PROCESS

Code assembly just consists of interpreting with the ASSEMBLER vocabulary as CONTEXT. Thus, each word in the input stream will be matched according the Forth practice of searching CONTEXT first then CURRENT.

```
ASSEMBLER     (now CONTEXT)
FORTH         (chained to ASSEMBLER)
user's vocab  (CURRENT if one exits)
FORTH         (chained to user's vocab)
try for literal number
else, do error abort
```

The above sequence is the usual action of Forth's text interpreter, which remains in control during assembly.

During assembly of CODE definitions, Forth continues interpretation of each word encountered in the input stream (not in the compile mode). These assembler words specify operands, address modes, and op-codes. At the conclusion of the CODE definition a final error check verifies correct completion by "unsmudging" the definitions name, to make it available for dictionary searches.

## RUN-TIME, ASSEMBLY-TIME

One must be careful to understand at what time a particular word definition executes. During assembly, each assembler word interpreted executes. Its function at that instant is called 'assembling' or 'assembly-time'. This function involves may involve op-code generation generation, address calculation, mode selection, etc.

The later execution of the generated code is called 'run-time'. This distinction is particulary important with the conditionals. At assembly time each such word (i.e. IF, UNTIL, BEGIN, etc.) itself 'runs' to produce machine code which will later execute at what is labeled 'run-time' when its named code definition is used.

## AN EXAMPLE

As a practical example, here's a simple call to the sytem monitor, via the NMI address vector (using the BRK opcode).

```
     CODE MON  ( exit to monitor )
               BRK,  NEXT JMP,    END-CODE
```

The word CODE is first encountered, and executed by Forth. CODE builds the following name "MON" into a dictionary header and calls ASSEMBLER as the CONTEXT vocabulary.

The "(" is next found in FORTH and executed to skip till ")". This method skips over comments. Note that the name after CODE and the ")" after "(" must be on the same text line.

## OP-CODES

BRK, is next found in the assembler as the op-code. When BRK, executes, it assembles the byte value 00 into the dictionary as the op-code for "break to monitor via "NMI".

Many assembler word names end in ",". The significance of this is:

1. The comma shows the conclusion of a logical grouping that would be one line of classical assembly source code.
2. "," compiles into the dictionary; thus a comma implies the point at which code is generated.
3. The "," distinguishes op-codes from possible hex numbers ADC and ADD.

## NEXT

Forth executes your word definitions under control of the address interpreter, named NEXT. This short code routine moves execution from one definition, to the next. At the end of your code definition, you must return control to NEXT or else to code which returns to NEXT.

## RETURN OF CONTROL

Most 6502 sytems can resume execution after a break, since the monitor saves the CPU register contents. Therefore, we must return control to Forth after a return from the monitor. NEXT is a constant that specifies the machine address of Forth's address interpreter (say $0242). Here it is the operand for JMP,. As JMP, executes, it assembles an machine code jump to the address of NEXT from the assembly time stack value.

## SECURITY

Numerous tests are made within the assembler for user errors:

1. All parameters used in CODE definitions must be removed.
2. Conditionals must be properly nested and paired.

3. Address modes and operands must be allowable for the op-codes.

These tests are accomplished by checking the stack position (in CSP) at the "smudged" creation of the definion name remains in the "smudged" conditon and will not be found during dictionary searches.

The user should be aware that one error not trapped is referencing a definition in the wrong vocabulary:

i.e. 0= of ASSEMBLER when you want
     0= of FORTH

SUMMARY

The object code of our example is:

```
3059   83 4D 4F CE      CODE MON
305D   4D 30            link field
305F   61 30            code field
3061   00               BRK
3062   4C 42 02         JMP NEXT
```

OP-CODES, revisited

The bulk of the assembler consists of dictionary entries for each op-code. The 6502 one mode op-codes are:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| BRK, | CLC, | CLD, | CLI, | CLV, | DEX, | DEY, | INX, |
| INY, | NOP, | PHA, | PHP, | PLA, | PLP, | RTI, | RTS, |
| SEC, | SED, | SEI, | TAX, | TAY, | TSX, | TXS, | TXA, |
| TYA, | | | | | | | |

When any of these are executed, the corresponding op-code byte is assembled into the dictionary.

The multi-mode op-codes are;

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ADC, | AND, | CMP, | EOR, | LDA, | ORA, | SBC, | STA, |
| ASL, | DEC, | INC, | LSR, | ROL, | ROR, | STX, | CPX, |
| CPY, | LDX, | LDY, | STY, | JSR, | JMP, | BIT, | |

These usually take an operand, which must already be on the stack. An address mode may also be specified. If none is given, the op-code uses z-page or absolute addressing. The address modes are determined by:

| SYMBOL | MODE | OPERAND |
|---|---|---|
| .A | accumulator | none |
| # | immediate | 8 bits only |
| ,X | indexed X | z-page or absolute |
| ,Y | indexed Y | z-page or absolute |

|  |  |  |
|---|---|---|
| X) | indexed indirect X | z-page only |
| )Y | indirect indexed Y | z-page only |
| ) | indirect | absolute only |
| none | memory | z-page or absolute |

EXAMPLES

Here are examples of Forth vs. conventional assembler. Note that the operand comes first, followed by any mode modifier, and then the op-code nmemonic. This makes best use of the stack at assembly time. Also, each assembler word is set off by blanks, as is required for all Forth source text.

```
         .A ROL,          ROL A          (.A distinguishes from
        1 # LDY,          LDY #1              hex number 0A)
    DATA ,X STA,          STA DATA,X
    DATA ,Y CMP,          CMP DATA,Y
       6 X) ADC,          ADC (06,X)
   POINT )Y STA,          STA (POINT),Y
  VECTOR  ) JMP,          JMP (VECTOR)
```

The words DATA and VECTOR specify machine addresses. In the case of "6 )X ADC," the operand memory address $0006 was given directly. This is occasionaly done if the usage of a value doesn't justify devoting the dictionary space to a symbolic value.

## 6502 CONVENTIONS

### Stack Addressing

The data stack is located in z-page, usually addressed by "Z-PAGE,X". The stack starts near $009E and grows downward. The X index register is the data stack pointer. Thus, incrementing X by two removes a data stack value; decrementing X twice makes room for one new data stack value.

Sixteen bit values are placed on the stack according to the 6502 convention; the low byte is at low memory, with the high byte following. This allows "indexed, indirect X" directly off a stack value.

The bottom and second stack values are referenced often enough that the support words BOT and SEC are included. Using

```
        BOT LDA,   assembles   LDA (0,X)  and
        SEC ADC,   assembles   ADC (2,X)
```

BOT leaves 0 on the stack and sets the address mode to ,X. SEC leaves 2 on the stack also setting the address mode to ,X.

Here is a pictorial representation of the stack in z-page.

```
|            |
|------------|
|  sec high  |
|  sec low   |
|------------|
|  bot high  |
```

The 0 or 2 left by BOT or SEC is the base address above which the X register indexes. You may further modify this at assembly time to

```
ess at any byte in the data stack.                | bot low   | <== X offset
                                                  |-----------|  above $0000
```

Here is an example of code to "or" to the accumulator four bytes he stack:

```
BOT    LDA,           LDA (0,X)
BOT 1+ ORA,           ORA (1,X)
SEC    ORA,           ORA (2,X)
SEC 1+ ORA,           ORA (3,X)
```

To obtain the 14-th byte on the stack:    BOT 13 + LDA,

RN STACK

The Forth Return Stack is located in the 6502 machine stack in Page 1. tarts at $01FE and builds downward. No lower bound is set or checked as e 1 has sufficient capacity for all (non-recursive) applications.

By 6502 convention the CPU's S register points to the next free byte ow the bottom of the Return Stack. The byte order follows the convention low significance byte at the lower address.

Return stack values may be obatained by: PLA, PLA, which will l1 the low byte, then the high byte from the return stack. To operate arbitrary bytes, the method is:

1) save X in XSAVE

2) execute TSX, to bring the S register to X.

3) use RP) to address the lowest byte of the return stack. Offset the value to address higher bytes. (Address mode is automatically set to ,X.)

4) Restore X from XSAVE.

As an example, this definition non-destructively tests that the second m on the return stack (also the machine stack) is zero.

```
CODE IS-IT    ( zero ? )
    XSAVE STX,  TSX,    ( setup for return stack )
    RP) 2+ LDA,  RP) 3 + ORA,   ( or 2nd item's two bytes together)
  0= IF,  INY,  THEN,    ( if zero, bump Y to one )
    TYA,  PHA,  XSAVE LDX,  ( save low byte, restore data stack)
    PUSH JMP,  END-CODE (  push boolean )
```

```
                 |Return Stack|
                 |------------|
                 |  hi byte   |     second item
                 |  lo byte   |
                 |------------|
                 |  hi byte   |     bottom item
R) =$0101,X -->  |  lo byte   |
                 |------------|
           S -->   free byte
```

## FORTH REGISTERS

Several Forth registers are available only at the assembly level and have been given names that return their memory addresses. These are:

| | |
|---|---|
| IP | address of the Interpretive Pointer, specifying the next Forth address which will be interpreted by NEXT. |
| W | address of the pointer to the code field of the dictionary definition just interpreted by NEXT. W-1 contains $6C, the op-code for indirect jump. Therefore, jumping to W-1 will indirectly jump via W to the machine code for the definition. |
| UP | User Pointer containing address of the base of the user area. |
| N | a utility area in z-page from N-1 thru N+7. |

## CPU Registers

When Forth execution leaves NEXT to execute a CODE definition, the following conventions apply:

1. The Y index register is zero. It may be freely used.
2. The X index register defines the low byte of the bottom data stack item relative to machine address $0000.
3. The CPU stack pointer S points one byte below the low byte of the bottom return stack item. Executing PLA, will pull this byte to the accumulator.
4. The accumulator may be freely used.

p

5. The processor is in the binary mode and must be returned in that mode.

## XSAVE

XSAVE is a byte buffer in z-page, for temporary storage of the X register. Typical usage, with a call which will change X, is:

```
CODE DEMO
    XSAVE STX,   USER'S JSR,  ( which will change X )
    XSAVE LDX,   NEXT JMP,   END-CODE
```

## N Area

When absolute memory registers are required, use the 'N Area' in the base page. These registers may be used as pointers for indexed/indirect addressing or for temporary values. As an example of use, see CMOVE in the system source code.

The assembler word N returns the base address (usually $00D1). The N Area spans 9 bytes, from N-1 thru N+7. Conventionally, N-1 holds one byte and N, N+2, N+4, N+6 are pairs which may hold 16-bit values. See SETUP for help on moving values to the N Area.

It is very important to note that many Forth proceedures use N. Thus N may only be used within a single code definition. Never expect that a value will remain there, outside a single definition!

```
CODE DEMO    HEX
     6 # LDA,  N 1 - STA,   ( setup a counter )
BEGIN,  8001 BIT,           ( tickle a port )
          N 1 - DEC,        ( decrement the counter )
       0< UNTIL,  NEXT JMP,  END-CODE    (  loop till negative )
```

SETUP

Often we wish to move stack values to the N area. The sub-routine SETUP has been provided for this purpose. Upon entering SETUP the accumulator specifies the quantity of 16-bit stack values to be moved to the N area. That is, A may be 1, 2, 3, or 4 only:

```
3 # LDA,  SETUP JSR,
```

```
       stack before        N after        stack after
            H high                               H
            G_low                      bot-->    G_
            F                   F
            E_                  E_
            D                   D
     sec--> C_                  C_
            B                   B
     bot--> A_          N-->    A_
```

CONTROL FLOW

Forth discards the usual convention of assembler labels. Instead, two replacements are used. First, each Forth definition name is permanently included in the dictionary. This allows proceedures to be located and executed by name at any time as well as be compiled within other definitions.

Secondly, within a code definition, execution flow is controlled by label-less branching according to "structured programming". This method is identical to the form used in colon-definitions. Branch calculations are done at assembly time by temporary stack values placed by the control words:

```
BEGIN,  UNTIL,  IF,  ELSE,  THEN,
```

Here again, the assembler words end with a comma, to indicate that code is being produced and to clearly differentiate from the high-level form.

One major difference occurs! High-level flow is controlled by run-time boolean values on the data stack. Assembly flow is instead controlled by processor status bits. The programmer must indicate which status bit to test, just before a conditional branching word (IF, and UNTIL,).

Examples are:

```
    PORT LDA,   0= IF,  <a>   THEN,
      (read port,  if equal to zero do  <a> )

    PORT LDA,   0= NOT IF,  <a>   THEN,
      (read port,  if not equal to zero do  <a> )
```

The conditional specifiers for 6502 are:

| | | | |
|---|---|---|---|
| CS | test carry set | C=1 | in processor status |
| 0< | byte less than zero | N=1 | |
| 0= | equal to zero | Z=1 | |
| CS NOT | test carry clear | C=0 | |
| 0< NOT | test positive | N=0 | |
| 0= NOT | test not equal zero | Z=0 | |

The overflow status bit is so rarely used, that it is not included. it is desired, compile:

```
  ASSEMBLER  DEFINITIONS   HEX
  50  CONSTANT  VS      ( test overflow set )
```

## ITIONAL LOOPING

A conditional loop is formed at assembler level by placing the ion to be repeated between BEGIN, and UNTIL,:

```
       6 # LDA,   N STA,          ( define loop counter in N )
  BEGIN, - PORT DEC,              ( repeated action )
           N DEC,     0= UNTIL,   ( N reaches zero )
```

First, the byte at address N is loaded with the value 6. The beginning he loop is marked (at assembly time) by BEGIN,. Memory at PORT is emented, then the loop counter in N is decremented. Of course, the CPU tes its status register as N is decremented. Finally, a test for Z=1 is e; if N hasn't reached zero, execution returns to BEGIN,. When N reaches (after executing PORT DEC, 6 times) execution continues ahead after IL,. Note that BEGIN, generates no machine code, but is only an assembly e locator.

## ITIONAL EXECUTION

Paths of execution may be chosen at assembly in a similar fashon at e in colon-definitions. In this case the branch is chosen based on a essor status condition code.

```
  PORT LDA,   0= IF,   <for zero set>   THEN,  <continuing code>
```

In this example, the accumulator is loaded from PORT. The zero status ested if set (Z=1). If so, the code <for zero set> is executed. er the zero status is set or not, exeucution will resume at THEN,.

The conditional branching also allows a specific action for the e case. Here we see the addition of the ELSE, part.

```
     PORT LDA,  0= IF,   <for zero set>
                      ELSE,  <for zero clear>
                      THEN,   <continuing code>
```

The test of PORT will select one of two execution paths, before resuming execution after THEN,. The next example increments N based on bit D7 of a port:

```
     PORT LDA,              ( fetch one byte )
      0< IF,  N DEC,        ( if D7=1, decrement N )
         ELSE,  N INC,      ( if D7=0, increment N )
         THEN,              ( continue ahead )
```

CONDITIONAL NESTING

Conditionals may be nested, according the the conventions of structured programming. That is, each conditional sequence begun (IF, BEGIN,) must be terminated (THEN, UNTIL,) before the next earlier conditional is terminated. An ELSE, must pair with the immediately preceeding IF,.

```
     BEGIN,    <code always executed>
        CS IF,  <code if carry set>
           ELSE,  <code if carry clear>   THEN,
        0= NOT UNTIL,  ( loop till condition flag is non-zero )
                <code that continues onward>
```

Next is an error that the assembler security will reveal.

```
     BEGIN,  PORT LDA,
        0= IF,  BOT INC,
               0= UNTIL,   ENDIF,
```

The UNTIL, will not complete the pending BEGIN, since the immediately preceeding IF, is not completed. An error trap will occur at UNTIL, saying "conditionals not paired".

RETURN OF CONTROL, revisited

When concluding a code definition, several common stack manipulations often are needed. These functions are already in the nucleus, so we may share their use just by knowing their return points. Each of these returns control to NEXT.

| | | |
|---|---|---|
| POP | POPTWO | remove one or two 16-bit stack values. |
| PUSH | | add two bytes to the data stack. |
| PUT | | write two bytes to the data stack, over the present bottom of the stack. |

Our next example complements a byte in memory. The bytes' address is on the stack when INVERT is executed.

```
      CODE INVERT  ( a memory byte )   HEX
          BOT X) LDA,    ( fetch byte addressed by stack )
           FF  # EOR,    ( complement accumulator )
          BOT X) STA,    ( replace in memory )
          POP JMP,   END-CODE ( discard pointer from stack, return to NEXT )
```

A new stack value may result from a code definition. We could program placing it on the stack by:

```
     CODE ONE                    ( put 1 on the stack )
          DEX,  DEX,             ( make room on the data stack )
          1 # LDA,  BOT STA,     ( store low byte )
                    BOT 1+ STY,  ( hi byte stored from Y since = zero )
                    NEXT JMP,  END-CODE
```

A simpler version could use PUSH:

```
      CODE ONE
         1 # LDA,  PHA,  ( push low byte to machine stack )
         TYA,  PUSH JMP,  ( high byte to accumulator, push to data stack )
         END-CODE
```

The convention for PUSH and PUT is:

1. push the low byte on to the machine stack.
2. leave the high byte in the accumulator.
3. jump to PUSH or PUT.

PUSH will place the two bytes as the new bottom of the data stack. PUT will over-write the present bottom of the stack with the two bytes. Failure to push exactly one byte on the machine stack will disrupt execution upon usage!

## FOOLING SECURITY

Occasionally we wish to generate unstructured code. To accomplish this we can control the assembly time security checks, to our purpose. First, we must note the parameters utilized by the control structures at assembly time. The notation below is taken from the assembler glossary. The --- indicates assembly time execution, and separates input stack values from the output stack values of the words execution.

```
    BEGIN,  ==>                      ---  addrB  1
    UNTIL,  ==>     addrB  1  cc  ---

    IF,     ==>               cc  ---  addrI  2
    ELSE,   ==>     addrI  2      ---  addrE  2
    THEN,   ==>     addrI  2      ---
              or    addrE  2      ---
```

The address values indicate the machine location of the corresponding 'B'EGIN, 'I'F, or 'E'LSE,. cc represents the condition code to select the processor status bit referenced. The digit 1 or 2 is tested for conditional pairing.

The general method of security control is to drop off the check digit and manipulate the addresses at assembly time. The security against errors is less, but the programmer is usually paying intense attention to detail during this effort.

To generate the equivalent of the high level:

```
BEGIN  <a>  WHILE  <b>  REPEAT
```

We write in assembly:

```
BEGIN,  DROP  ( the check digit 1, leaving addrB)
          <a>
  CS IF,     ( leaves addrI and digit 2 )
          <b>
     ROT  ( bring addrB to bottom )  JMP, ( to addrB  of BEGIN, )
    ENDIF,  ( complete false foward branch from IF, )
```

It is essential to write the assembly time stack on paper, and run through the assembly steps, to be sure that the check digits are dropped and re-inserted at the correct points and addresses are correctly available.

## ASSEMBLER GLOSSARY

#

Specify 'immediate' addressing mode for the next op-code generated.

)Y

Specify 'indirect indexed Y' addressing mode for the next op-code generated.

,X

Specify 'indexed X' addressing mode for the next op-code generated.

,Y

Specify 'indexed Y' addressing mode for the next op-code generated.

.A

Specify accumulator addressing mode for the next op-code generated.

0<          --- cc  (assembling)

Specify that the immediately following conditional will branch based on the processor status bit being negative (Z=1), i.e. less than zero. The flag cc is left at assembly time; there is no run-time effect on the stack.

0=          --- cc  (assembling)

Specify that the immediately following conditional will branch based on the processor status bit being equal to zero (Z=1). The flag cc is left at assembly time; there is no run-time effect on the stack.

;CODE

Used to conclude a colon-definition in the form:

```
    : <name>    . . .   ;CODE   <assembly code>    END-CODE
```

Stop compilation and terminate a new defining word <name>. Set the CONTEXT vocabulary to ASSEMBLER, assembling to machine code the following nmemonics. An existing defining word must exist in <name> prior to ;CODE.

When <name> later executes in the form:

```
        <name>  <namex>
```

the definition <namex> will be created with its execution proceedure given by the machine code following <name>. That is, when <namex> is executed, the address interpreter jumps to the code following ;CODE in <name>.

ASSEMBLER                                                  in FORTH

Make ASSEMBLER the CONTEXT vocabulary. It will be searched first when the input stream is interpreted.

```
BEGIN,        ---  addr  1     (assembling)
              ---              (run-time)
```

Occurs in a CODE definition in the form:

```
    BEGIN,  . . .  cc  UNTIL,
```

At run-time, BEGIN, marks the start of an assembly sequence repeatedly executed. It serves as the return point for the corr-esponding UNTIL,. When reaching UNTIL, a branch to BEGIN, will occur if the processor status bit given by cc is false; other-wise execution continues ahead.

At assembly time, BEGIN, leaves the dictionary pointer address addr and the value 1 for later testing of conditionary pairing by UNTIL,.

```
BOT             ---  n  (assembling)
```

Used during code assembly in the form:

```
            BOT  LDA,    or    BOT 1+ X) STA,
```

Addresses the bottom of the data stack (containing the low byte) by selecting the ,X mode and leaving n=0, at assembly time. This value of n may be modified to another byte offset into the data stack. Must be followed by a multi-mode op-code nmemonic.

CODE

A defining word used in the form:

```
     CODE <name>    . . . .   END-CODE
```

to create a dictionary entry for <name> in the CURRENT vocabulary. <name>'s code field contains the address of its parameter field. When <name> is later executed, the machine code in this parameter field will execute. The CONTEXT vocabulary is made ASSEMBLER, to make available the op-code nmemonics.

```
CPU          n  ---   (compiling assembler)
```

An assembler defining word used to create assembler nmemonics that have only one addressing mode:

```
            EA  CPU  NOP,
```

CPU creates the word NOP, with its op-code EA as a parameter. When NOP, later executes, it assembles EA as a one byte op-code.

```
CS              ---  cc  (assembling)
```

Specify that the immediately following conditional will branch based on the processor carry is set (C=1). The flag cc is left at assembly time; there is no run-time effect on the stack.

```
ELSE,                          ---             (run-time)
                addr1  2  ---  addr2  2  (assembling)
           Occurs within a code definition in the form:
                   cc  IF,  <true part>  ELSE,  <false part>  THEN,
           At run-time, if the condition code specified by  cc  is false,
           execution will skip to the machine code following  ELSE,.
           At assembly time  ELSE,  assembles a forward jump to just after
           THEN,  and resolves a pending forward branch from IF.  The values
           2 are used for error checking of conditional pairing.

END-CODE
           An error check word marking the end of a CODE definition.
           Successful execution to and including  END-CODE  will unsmudge the
           most recent  CURRENT  vocabulary definition, making it available
           for execution.  END-CODE also exits the ASSEMBLER making CONTEXT the
           same as CURRENT.  This word previously was named  C;

IF,                cc  ---  addr  2  (assembly-time )
                       ---  addr  2   (assembly-time)
           Occurs within a code definition in the form:
                   cc  IF,  <true part>  ELSE,  <false part>  THEN,
           At run time,  IF, branches based on the condition code cc,
           (0< or 0= or CS).  If the specifed processor status is true,
           execution continues ahead, otherwise branching occurs to just
           after  ELSE,  (or  THEN, when  ELSE, is not present).
           At ELSE, execution resumes at the corresponding THEN,.

           When assembling,  IF, creates an unresolved forward branch based
           on the condition code cc, and leaves addr and 2 for resolution
           of the branch by the corresponding ELSE, or THEN,.  Conditionals
           may be nested.

INDEX             ---  addr   (assembling)
           An array used within the assembler, which holds bit patterns of
           allowable addressing modes.

IP                ---  addr   (assembling)
           Used in a code definition in the form:
                     IP STA,    or    IP )Y LDA,
           A constant which leaves at assembly time the address of the pointer
           to the next Forth execution address in a colon-definition to be
           interpreted.

           At run-time,  NEXT moves IP ahead within a colon-definition.
           Therefore,  IP points just after the execution address being
           interpreted.  If an in-line data structure has been compiled (i.e.
           a character string), indexing ahead by  IP  can access this data:

                     IP STA,    or    IP )Y LDA,
           loads the third byte ahead in the colon-definition being interpreted.

M/CPU           n1  n2  ---   (compiling assembler)
           An assembler defining word used to create assembler nmemonics that
           have multiple address modes:
                  1C6E  60  M/CU  ADC,
```

M/CPU creates the word ADC, with two parameters. When ADC, later executes, it uses these parameters, along with stack values and the contents of MODE to calculate and assemble the correct op-code and operand.

I Used within the assembler to set MODE to the default value for direct memory addressing, z-page.

E --- addr
A variable used within the assembler, which holds a flag indicating the addressing mode of the op-code being generated.

N --- addr (assembling)
Used in a code definition in the form:

```
        N 1 - STA,     or     N 2+ )Y ADC,
```

A constant which leaves the address of a 9 byte workspace in z-page. Within a single code definition, free use may be made over the range N-1 thru N+7. See SETUP.

T ---- addr (assembling)
A constant which leaves the machine address of the Forth address interpreter. All code definitions must return execution to NEXT, or code that returns to NEXT (i.e. PUSH, PUT, POP, POPTWO).

T cc1 --- cc2 (assembly-time)
When assembling, reverse the condition code for the following conditional. For example:

```
        0= NOT IF, <true part> THEN,
```

will branch based on 'not equal to zero'.

P --- addr (assembling)
n --- (run-time)
A constant which leaves (during assembly) the machine address of the return point which, at run-time, will pop a 16-bit value from the data stack and continue interpretation.

O --- addr (assembling)
n1 n2 --- (run-time)
A constant which leaves (during assembly) the machine address of the return point which, at run-time, will pop two 16-bit values from the data stack and continue interpretation.

H --- addr (assembling)
--- n (run-time)
A constant which leaves (during assembly) the machine address of the return point which, at run-time, will add the accumulator (as high-byte) and the bottom machine stack byte (as low-byte) to the data stack.

T --- addr (assembling)
n1 --- n2 (run-time)
A constant which leaves (during assembly) the machine address of the return point which, at run-time, will write the accumulator (as high-byte) and the bottom machine stack byte(as low-byte) over the existing data stack 16-bit value (n1).

```
RP)             ---    (assembly-time)
          Used in a code definition in the form:
                    RP) LDA,    or   RP) 3+ STA,
          Address the bottom byte of the return stack (containing the low byte)
          by selecting the  ,X mode  and leaving  n=$101.  n may be modified
          to another byte offset.  Before operating on the return stack
          the X register must be saved in XSAVE and  TSX, be executed;
          before returning to NEXT,  the X register must be restored.

SEC            --- n  (assembling)
          Identical to  BOT,  execpt that  n=2.  Addresses the low byte of
          the second 16-bit data stack value (third byte on the data stack).

THEN,                        ---   (run-time)
                 addr  2  ---   (assembly-time)
          Occurs in a code definition in the form:
                 cc  IF,  <true part>  ELSE,  <false part>  THEN,
          At run-time THEN, marks the conclusion of a conditional structure.
          Execution of either the true part or false part resumes following
          THEN,.  When assembling  addr  and  2  are used to resolve the
          pending forward branch to THEN,.

UNTIL,                        ---   (run-time)
             addr  1  cc   ---   (assembling)
          Occurs in a CODE  definition in the form:
                 BEGIN,  . . .  cc  UNTIL,
          At run-time,  UNTIL,  controls the conditional branching back to
          BEGIN,.  If the processor status bit specified by cc is false,
          execution returns to BEGIN,;  otherwise execution continues ahead.

          At assembly time,  UNTIL,  assembles a conditional relative branch
          to  addr  based on the condition code  cc.  The number 1 is used for
          error checking.

UP              ---  addr    (assembling)
          Used in a code definition in the form:
                   UP LDA,    or  UP )Y STA,
          A constant leaving at assembly time the address of the pointer to
          the base of the user area.  i.e.
                HEX    12 # LDY,    UP )Y LDA,
          load the low byte of the sixth user variable, DP.

UPMODE       addr  f  ---  addr  f
          Used within the assembler to adjust the addressing mode based on
          the operand size and opcode type.

W               ---  addr    (assembling)
          Used in a code definition in the form:
                 W 1+ STA,    or    W 1 - JMP,    or   W )Y ADC,
          A constant which leaves at assembly time the address of the pointer
          to the code field (execution address) of the Forth dictionary word
          being executed.  Indexing relative to  W  can yield any byte in the
          definitions parameter field.  i.e.
                 2 # LDY,   W )Y LDA,
          fetches the first byte of the parameter field.
```

X)

Specify 'indexed indirect X' addressing mode for the next op-code generated.

XSAVE --- addr (assembling)

Used in a code definition in the form:

```
      XSAVE STX,   or  XSAVE LDX,
```

A constant which leaves the address at assembly time of a temporary buffer for saving the X register. Since the X register indexes to the data stack in z-page, it must be saved and restored when used for other purposes.

OK